A Formal IT-Security Model for the Correction and Abort Requirement of Electronic Voting[1]

Rüdiger Grimm[1], Katharina Hupf[1], and Melanie Volkamer[2]

[1]Institute of Information Systems Research
Universität Koblenz-Landau
Universitätsstraße 1
56070 Koblenz
Germany
{grimm,hupfi}@uni-koblenz.de

[2]Center for Advanced Security Research Darmstadt
Technische Universität Darmstadt
Mornewegstraße 32
64293 Darmstadt
Germany
volkamer@cased.de

Abstract: This paper addresses a basic security requirement of electronic voting, namely that a voter can correct or abort his vote at any time prior to his final vote casting. This requirement serves as a protection against voter precipitance (haste). We specify rules for a reset and cancel function that implement the correction and abort requirement. These rules are integrated in an extended version of the formal IT security model provided in [VG08]. We show that these rules do respect the requirements covered in this model namely that each voter can cast a vote, that no voter loses his voting right without having cast a vote and that only eligible voters can cast a vote. This paper formally describes and mathematically proves the model and finally shows at which places of a voting process the formal rules apply.

[1] This paper is developed within the project "ModIWA – Modellierung von Internetwahlen" which is funded by DFG, and carried out at the Universities Kassel (Roßnagel, Richter) and Koblenz-Landau (Grimm, Hupf)

1 Introduction

Security is an elementary property of electronic voting systems and is thus fundamental for the trust of the voters in the system. Security objectives for electronic voting were first collected in an informal way, for example by a European-wide accepted recommendation adopted by the Council of Europe [CE04]. Later the semi-formal method of the Common Criteria [CC06] was used to specify a Protection Profile (PP) for a basic set of security requirements for online voting products [VV08]. There are good reasons to specify the security objectives of an IT system in a formal way, i.e., by mathematical calculus which states and proves properties clearly [Wa05]. The formalization of security objectives is a way to gain unambiguous and clearly understood

Requirements for electronic voting. Due to its formal base, it can be mathematically proven that a specification or implementation conforms to these formal security requirements. For example, the mandatory access model of Bell and LaPadula [BP73] strengthens the trust in a secure centrally controlled multi-user computer system, such that in the early days of computer system security evaluation it used to define the highest assurance level of the Orange Book Criteria [DD85]. Thus a formal IT security model on electronic voting defining security requirements from [CE04] and [VV08] in a formal language can create large amounts of trust in the effect of the security functions implemented in the electronic voting system.

However, the Common Criteria Protection Profile for online voting products [VV08] requires an evaluation according to evaluation assurance level EAL2+ on a scale from 1 to 7. This level does not require any formal proof. This evaluation level seems to be acceptable as this PP only claims to define basic requirements. Parliamentary elections, however, demand a higher evaluation level, probably EAL 6 or 7. At this level, the application of formal methods and the definition of a formal security model [CC06] are mandatory for the Common Criteria evaluation.

To enable a Common Criteria evaluation according to these levels, the authors of [VG08] provide an IT security sub-model for electronic voting. However, this model only covers a small subset of security objectives namely that each voter can cast a vote, that no voter loses his voting right without having cast a vote and that only eligible voters can cast a vote. This model needs to be extended to meet the remaining security objectives. The aim of this paper is to extend the protection against errors by haste (precipitation). Moreover, in extending the model in [VG08] we have found a weakness in the model which is corrected in this paper, as well.

Protection against errors by haste is a basic legal requirement well established in private and public law [Ba06]. This requirement is expressed by two security objectives in [VV08], “O.Correction” and “O.Abort,” as well as by the security objectives 10 and 11 in [CE04]. To meet these two security objectives, we will propose two functions “reset” and “cancel” of a voting process. The abortion of a voting process protects not only against precipitation, but it also protects the secrecy of voting against unwanted external events like the appearance of another person during the voting process. Thus reset and cancel are important for the support of the freedom of vote.

The paper is organized as follows: In the subsequent section 2 we quote those security objectives, from the Protection Profile on basic requirements for online voting products [VV08], that we are going to formalize in this paper. In section 3 we enhance the existing formal IT security model in [VG08] according to our findings and provide a full proof of its correctness. In section 4 we formalize the "reset" and "cancel" functions, which have been introduced in section 2. In section 5 we prove that this extended security model is correct and, thus, provides a smooth extension of the original security model [VG08]. To complete the picture, in section 6 we show (informally) at which points in a voting process our security rules of the formal model are applied. Finally, in section 7 we draw some conclusions from our work and point to further research.

2 Security Objectives

Security models start with the identification of security objectives [CC06, Gr08]. In the protection profile of a basic set of security requirements for online voting [VV08], a set of thirty-two security objectives for online voting products are specified. The following two of these have been used as a first step towards a formal model for remote electronic voting systems in [VG08][2]:

O.OneVoterOneVote: It is ensured that (a) each voter can cast one vote and (b) no voter loses his voting right without having cast a vote.

O.UnauthVoter: Only eligible voters who are unmistakably identified and authenticated are allowed to cast a vote that is stored in the ballot box.

These two objectives are met by specifying properties that define "secure system states" and rules to be applied on any function that securely transfers a system state into another system state. Therefore, these rules are called transition rules. After specifying the related security state properties and transition rules of these two security objectives, we will extend the model by including two more security objectives from [VV08], namely:

O.Abort: The voter can abort his voting process at any time prior to the final casting of the vote without loosing his right to vote.

O.Correction: There is no limit on the number of corrections a voter can make to his vote until the final casting of the vote.

These objectives will not be met by security properties, but by a further transition rule. We propose that "reset" and "cancel" functions are the appropriate prototype functions of this rule, whereby "cancel" will be a repetition of "reset" until the initial state of a voter's voting process. We will prove (in section 5) that these rules preserve the security properties of **O.OneVoterOneVote** and **O.UnauthVoter**.

[2] We refer to [VV08] as well. This paper formally models some basic security requirements for electronic voting, which apply to both voting machines and online voting.

The rules for allowed state transitions are to be implemented by voting products as functions for data processing. However, the rules do not determine appropriate places for these functions within a voting process. Strictly speaking, it is not the purpose of an IT security model to design processes or protocols. Although we are not going to design the voting process, we will show (in section 6) informally at which points in a voting process our rules (and especially the "reset" and "cancel" functions) would be applied.

3 The Basic Model

3.1 The original model of [VG08]

We quote the basic model from [VG08] in that we take the security objectives **O.OneVoterOneVote** and **O.UnauthVoter** and associate them with properties of a secure state and allowed state transitions. Before we define the security properties, we define (general) system states of a voting process:

Definition 1 (voting system state)

A system state $S := <W, V, voter>$ is represented by a triple of the following three entries:

1. W – Set of eligible voters (those who are listed in the electoral register and have not yet cast a vote).
2. V – Set of (encrypted) votes stored in the e-ballot box.
3. $voter: V \rightarrow M$ – Mapping of (encrypted) votes to their electors.

W_{total} is the set of all eligible voters registered by the responsible voting officials before the voting system is started. M is a superset of W_{total} that contains any user who tries to access the remote electronic voting system, whether or not this particular user has the right to cast a vote. The function *voter* assigns each (encrypted) vote to its producer (voter).

The initial state is defined as the triple $S_0 := < W_{total}, V_0=\{\}, voter_0=\{\}>$.

We assume that state transitions t_1, t_2 … that carry the system from state to state are stimulated by events such as the login of a voter into the system, the request of an empty voting ballot, the filling out of the ballot, the casting of a vote, etc.

$$S_0 \xrightarrow{t_1} S_1 \xrightarrow{t_2} \ldots \xrightarrow{t_i} S_i$$

Now we follow the basic model in [VG08] and proceed to defining secure system states, and then we state the rules for allowed state transitions.

Definition 2 (secure voting system state, basic version)

A state S_i is a secure state iff (all of) the following constraints hold:

$OneVoterOneVote\ (A)$	$\forall v, v' \in V_i : voter(v) = voter(v') \Rightarrow v = v'$
$OneVoterOneVote\ (B)$	$\forall w \in W_{total} \setminus W_i : \exists v \in V_i : voter(v) = w$
$UnauthVoter$	$\forall v \in V_i : voter(v) \in W_{total}$

Definition 3 (rules for permitted state transitions)

A state transition from state S_i to state S_{i+1} stimulated by event t_{i+1} is permitted, $permitted\left(S_i \xrightarrow{t_{i+1}} S_{i+1}\right)$, if one of the following rules holds:

[Rule 1] $W_i = W_{i+1} \wedge V_i = V_{i+1} \wedge voter_i = voter_{i+1}$

[Rule 2] $\exists v \in V_{i+1} : (voter_{i+1}(v) \in W_i \wedge W_{i+1} = W_i \setminus \{voter_{i+1}(v)\} \wedge V_i = V_{i+1} \setminus \{v\}\ )$

[Rule 1] represents a state transition in which no vote is cast whereas [Rule 2] models a state transition during which an eligible voter casts a vote into the ballot box. This voter is eliminated from the list of eligible voters and his vote is stored in the ballot box.

3.2 Discussion of the original model

The security theorem in [VG08] proves that "for all permitted state transitions starting with the initial state [...] holds that any reachable state is secure." This security theorem is correctly proven. But it doesn't regard those secure states that are reached by an illegal state transition. Any state reachable by a permitted state transition from a secure state is obliged to be secure, even if the initial state (which is secure) has been reached for any reason by a non-permitted state transition. The following example shows that this isn't fulfilled for the formal security model in [VG08]:

Assume an eligible voter casts a vote into the ballot box, but –due to erroneous system implementation– the voter isn't eliminated from the list of eligible voters. The succeeding system state remains secure because *OneVoterOneVote(B)* doesn't specify properties of W_i, but only of $W_{total} \setminus W_i$. Suppose this voter casts a vote again. Since this voter is still eligible, his vote is stored in the ballot box and he is eliminated from the list of eligible voters. This represents a permitted state transition according to [Rule 2]. But the ballot box now contains two votes from the same voter. Thus an insecure system state is reached from a secure state by a permitted state transition.

To avoid this situation the definition of secure states needs to be extended such that a voter who has cast a vote into the ballot box is removed from the list of eligible voters. This can be incorporated into the formal model of [VG08] by extending definition 2 by an additional requirement for secure states:

$$OneVoterOneVote\ (C) \qquad \forall w \in W_i : \forall v \in V_i : voter(v) \neq w$$

Still, this extension isn't sufficient yet. Let S_i be a secure state. Furthermore, assume that an eligible voter *x* who hasn't yet cast a vote wants to vote. Let the system be in a state where the voter's eligibility is provable, i.e., $x \in W_i$. Due to an incomplete or incorrect list of registered voters, let $x \notin W_{total}$. This situation and $x \in W_i \setminus W_{total}$ are not forbidden by the definition of a secure state. Therefore, the system would follow [Rule 2] and let *x* cast a vote *v*, such that $V_{i+1} = V_i \cup \{v\}$ holds. Even though state S_i was secure and the state transition from S_i to S_{i+1} was permitted, state S_{i+1} isn't secure since $x = voter(v) \notin W_{total}$ violates the security property *UnauthVote*.

To avoid this situation, we add one more requirement for secure states, namely, that the system allows only registered voters ($x \in W_{total}$) to cast a vote ($x \in W_i$):

$$EligibleVoters \qquad W_i \subseteq W_{total}$$

This leads our enhanced security model's definition of a secure state.

3.3 The enhanced model

We now include the additional security properties *OneVoterOneVote* (*C*) and *EligibleVoters* from our discussion in section 3.2 above to the three security properties *OneVoterOneVote* (*A* and *B*) and *UnauthVoter* from definition 2 in section 3.1 above and thus we get the final definition of a secure state by these five security properties:

Definition 4 (secure voting system state, advanced version)

A voting system state S_i is a secure state if (all of) the following constraints hold:

$OneVoterOneVote\ (A)$	$\forall v, v' \in V_i : voter(v) = voter(v') \Rightarrow v = v'$
$OneVoterOneVote\ (B)$	$\forall w \in W_{total} \setminus W_i : \exists v \in V_i : voter(v) = w$
$OneVoterOneVote\ (C)$	$\forall w \in W_i : \forall v \in V_i : voter(v) \neq w$
$EligibleVoters$	$W_i \subseteq W_{total}$
$UnauthVoter$	$\forall v \in V_i : voter(v) \in W_{total}$

Obviously, the five properties above are equivalent to the two following properties:

(ap.1) *voter* is an injective function (equivalent to *OneVoterOneVote* (*A*)),

(ap.2) $W_{total}=W_i+voter(V_i)$ ("direct sum", equivalent to the other four properties). The direct sum means that both hold, $W_i \cup voter(V_i)=\mathrm{W}_{total}$, and $W_i \cap voter(V_i)=\varnothing$.

The proof that (ap.1) and (ap.2) are equivalent to definition 4 is straight forward and left as an exercise to the reader. It is also easy to see that the initial state S_0 is secure, because the voter function is empty, and hence injective; and $W_0 \cup voter(V_0)=\mathrm{W}_{total} \cup \varnothing=\mathrm{W}_{total}$ *; and* $W_0 \cap voter(V_0)= \mathrm{W}_{total} \cap \varnothing=\varnothing$.

Security Theorem

Permitted state transitions of definition 3 carry secure states into secure states according to definition 4.

Proof: In [VG08] we proved the security theorem in the weaker version that starting with S_0 any sequence of allowed state transitions would always lead to a secure state. We had to prove this by mathematical induction. Here we prove a stronger version that starting from any secure state (regardless of how this state was reached) an allowed state transition according to [Rule 1] or [Rule 2] will always reach a secure state. That is, we have to prove directly: For any $i \geq 0$, if we assume that S_i is secure, i.e., it has properties (ap.1) and (ap.2), and that $permitted\left(S_i \xrightarrow{t_{i+1}} S_{i+1}\right)$, i.e., t_{i+1} follows [Rule 1] or [Rule 2], then we have to show that properties (ap.1) and (ap.2) also hold for S_{i+1}.

Let t_{i+1} follow [Rule 1]. Then $V_{i+1}= V_i$ and $W_{i+1}= W_i$ and $voter_{i+1}= voter_i$, thus S_{i+1} simply inherits the security properties (ap.1) and (ap.2) from S_i.

Let t_{i+1} follow [Rule 2]. Then exactly one eligible voter casts a vote *v* into the ballot box during state transition t_{i+1}. Thus, $W_{i+1}= W_i \setminus \{voter_{i+1}(v)\}$ and $V_{i+1}= V_i \cup \{v\}$ holds.

(ap.1) Then $voter_{i+1}$ is injective on $V_i \cup \{v\}$, because $voter_{i+1}$ restricted on V_i is, by definition, equal to $voter_i$, which is injective, and $voter_{i+1}(v)$ does not match with any other image of $voter_i$, because $voter_{i+1}(v) \in W_i \setminus W_{i+1} \subset W_i$ and hence cannot have been in $voter_i(V_i)$ since $W_i \cap voter(V_i) = \emptyset$.

(ap.2) *(i)* $W_{i+1} \cup voter(V_{i+1}) = (W_i \setminus \{voter(v)\}) \cup voter(V_i \cup \{v\}) = (W_i \setminus \{voter(v)\}) \cup (voter(V_i) \cup \{\mathrm{voter}(v)\}) = W_i \cup voter(V_i) = W_{total}$. The last equality holds because S_i has property (ap.2).

(ii) $W_{i+1} \cap voter(V_{i+1}) = (W_i \setminus \{voter(v)\}) \cap voter(V_i \cup \{v\}) = (W_i \setminus \{voter(v)\}) \cap (voter(V_i) \cup \{\mathrm{voter}(v)\}) = W_i \cap voter(V_i) = \emptyset$. The last equality holds because S_i has property (ap.2).

4 An additional transition rule for "reset" and "cancel"

In this section we incorporate the security objectives *O.Abort* and *O.Correction* into the enhanced formal model. For this purpose we introduce an additional transition rule [Rule 3], which meets these objectives and will, therefore, be associated with a secure "reset" and "cancel" function.

4.1 Informal description of "reset" and "cancel"

While *O.Abort* is correlated with the sending and receiving of "cancel," *O.Correction* is associated with the sending and receiving of "reset." With "reset" we mean that during a voting process a voter can go back one step just before the last message that he sent to the server. With "cancel" we mean, that a voter can repeat reset events back to the initial state so that he can restart his individual voting process. On the receiving side, after a voter's "reset" the voting server must filter out all events that were stimulated by messages exchanged with this voter just before the last message received from this voter. However, all other events stimulated by messages with other voters must be kept by the voting server. On receiving a "cancel" message from a voter, the voting server must forget all events by messages exchanged with this voter, but keep all events stimulated by other voters. The sending and receiving of a "reset" and "cancel" message must be carefully synchronized between voters and their voting server. As a security rule, the "reset" must not create or delete voting rights or cast votes

4.2 Formal basics

The formalization of the "reset" and "cancel" functions requires some formal basics on lists and list operations and a communication function on events. Readers who are familiar with the formal specifications can skip to section 4.3.

Let M denote the set of all communication partners. Then we assume communication partners a, b, $\ldots \in M$ who observe events that are correlated by a communication function *com*. Partner a will be a model for a voter and partner b will be a model for a voting server. Each partner observes events on his side that are stimulated by the sending and receiving of messages. Events are communicated via messages. If a sends a message of type e to b, then a observes the event of type e that he sends to b, and b would observe this event with the label e as a message of type e that he receives from a. In the following we will use the terms "message" and "message type" with the same meaning as "event" and "event label", respectively. We will sometimes say, "sending event" and "receiving event" instead of "sent message" or "received message." The following event labels (=message types) are useful for the modeling of electronic voting, e.g., [VV08]. Note that they are just an example which we will take up in section 6. They are not exhaustive. For example, message types "error" or "verify" are ignored throughout this paper's model.

$$Eventlabels = \{\pm login, \pm requestBallot, \pm vote, \pm reset, \pm cancel, \pm confirmBallot, \pm castVote, \pm feedback, \pm logout\}$$

Let $e \in Eventlabels$ then $sig(e)$ denotes the algebraic sign of e. A negative sign of an event label e indicates that the associated event is being sent, e.g., $e = -login$. A positive sign indicates the associated event is being received, for example, $e = confirmBallot$.

Events are event labels associated with their sender and recipient. We denote the set of all possible events as

$$Events \subseteq Eventlabels \times M \times M$$

Let $a,b \in M$ and $e \in Eventlabels$. Events are defined as triples, but for convenience we will use the following notation for events instead (cf. [Gr09]):

$a(e:b)$	a receives a message e from b
$a(-e:b)$	a sends a message e to b

Let for $1 \leq k \leq n$ π_k denote the set-theoretic projection of a Cartesian product of n sets on its k-th component. Let $x = a(\pm e:b)$ be an event and π_i the projection of a tuple to its i-th element, then

$\pi_1(x)$ returns the event label e of x, which may carry a positive or negative sign.

$\pi_2(x)$ returns $a \in M$. Note that a is the sender of the message e if $sig(e)$ is positive, and a is the recipient if $sig(e)$ is negative.

$\pi_3(x)$ returns $b \in M$. Note that b is the recipient of the message e if $sig(e)$ is positive, and b is the sender if $sig(e)$ is negative.

For the synchronization of events that are stimulated by messages between *a* and *b*, we need a way to express that a message is observed by both sides. Let *Events* be the set of all possible events, $a,b \in M$ and $e \in Eventlabels$. The function *com* is defined as in [Gr09] and maps the sending and receiving of a message on the corresponding event on the partner's side:

$com: Events \rightarrow Events$

$com(a(e{:}b)) := b(-e{:}a)$

$com(a(-e{:}b)) := b(\mathrm{e}{:}a)$

We are going to collect events in ordered lists of events which allow us to operate on sequences of events and on identified events within the list. The algebra of ordered lists is a standard formalism used in theoretical computer science, see for example [MG08]. As usual, a list of events is understood as a finite sequence (or n-tupel) of these events. If *op* is a function on lists, for example the deletion of its head element, then the *k*-times repetition of the operation is denoted as $op^k(L) = op_k(op_{k-1}(...(op_1(L))...))$.

Useful list functions are [MG08]:

- For any list *L* of elements of a set *Q*, $set(L) \subset Q$ denotes the (unordered) set that consists of all elements of *L*.
- *head(L)* and *tail(L)* return the last element of *L*, and the rest of the list *L* without the last element, respectively. In contrast, $\overline{tail}$ is complementary to *tail* and returns the remaining list without the first element of *L*.
- Let $q \in Q$, then $L||q$ appends the element *q* at the end of the list *L*.
- $|L|$ returns the number of elements in *L*.
- Assume $n \in \mathrm{N}$ a natural number and $q \in Q$. $L[n]$ returns the element at the *n*-th position in the list and *pos*(*L,q*) returns the position of the last occurrence of the element *q* in the list *L*.
- *del*(*L,l*) with $l \in \mathrm{N}$ a natural number returns the list *L*, from that the *l*-th and all succeeding elements are removed.
- Especially for lists *L* of events, we define a filter function, a remove function and a select function. For an event *x* and $k \in \{1,2,3\}$, $filter_k(L,x)$ removes all events with event label *x* from the list *L* if *k*=1, or it removes all events whose first or second actor is *x* from the list *L* if *k*=2,3, and then returns the remaining list. For a communication partner *a, rmv*(*L,a*) returns the list *L* from which all events that were sent *or* received by *a* are removed. The function $select_k(L,x)$ returns the list of events where only those events with the event label *x* are contained if *k*=1*,* or only those events whose first or second actor is *x* are contained if *k*=2,3.

4.3 Formalized "reset" and "cancel"

We are now ready to formally define the "reset" and "cancel" event and prove the important synchronization theorem. For simplicity we assume in the following that *a* communicates solely with *b*, while *b* communicates with *a* and other partners as well. Thus in the model, *a* represents a voting client and *b* represents the voting server.

Definition 5 (Reset)

Let $a,b \in M$ and X_i be the list of events on the side of communication partner a , i.e., $\forall x \in X_i : \pi_2(x) = a$. Furthermore, let Y_j denote the list of events on the side of communication partner *b*. Let *sent(X_i)* denote the list of events that contains the send-events of X_i only, and let *received(Y_j)* denote the list of events that contains the receive-events of Y_j only, then we define:

$$X_i \| a(-reset : b) := \begin{cases} X_0 & if \quad set(sent(X_i)) = \varnothing \\ del(X_i, l) & else, where \quad l = \max\{ n \in N \mid x_n \in set(sent(X_i)\} \end{cases}$$

$$Y_j \| b(\,reset : a) := \begin{cases} del(Y_j, k) \,\|\, rmv(\overline{tail}^{k}(Y_j, a)) & if \quad C_2 \\ filter_3(Y_j, a) & else \end{cases}$$

where C_2 is $\exists k > 0 : k = \max\{n \in N \mid y_n \in set(received(filter_3(Y_j, a)))\}$

Explanation: If a communication partner $a \in M$ executes a "reset" then the last event $x_l \in X_i$ which is sent by *a* and all successive events to x_l are deleted. If there is no event in X_i that is sent by *a* (i.e., X_i is empty or contains only received events), then X_i is set to its initial state.

If a communication partner $b \in M$ receives a "reset" then the last event $y_k \in Y_j$ that is received by *b* from *a* is deleted as well as all successive events to y_k, which are sent to or received from *a* by *b*. Remark that all events successive to y_k, which are exchanged with other communication partners, are preserved in the state of communication partner *b*. If there is no event in Y_j that is received from *a* by *b* (i.e., Y_j is empty, doesn't contain any events exchanged with *a* or contains only events sent to *a*), then all messages sent by *b* to *a* are deleted from the list Y_j, i.e., *b* is set to initial state with respect to *a*. All events that are exchanged with different communication partners are preserved.

General Assumptions:

The reset and cancel functions are to be synchronized between voters and server. They wouldn't work properly if the system is interrupted. Therefore, availability is a security requirement for all communication functions. For the purpose of our security considerations, we assume that our systems are available and work correctly. Therefore, we assume secure communication channels in the following sense:

(A1) $\exists i \geq 0 : x \in set(X_i) \quad \Leftrightarrow \quad \exists j \geq 0 : com(x) \in set(Y_j)$

If a communication partner *a* exchanges a message *x* with *b* then there exists a state such that this message is observable on the partner's side.

(A2) $\forall i > 0 : \forall x_n, x_m \in set(sent(X_i)) : pos(X_i, x_n) < pos(X_i, x_m) \quad \Leftrightarrow$
$\forall j \geq i : pos(Y_j, com(x_n)) < pos(Y_j, com(x_m))$

If a communication partner *a* sends two messages in a particular order then the communication partner *b* receives them in exactly that order.

Theorem (Synchronization property of "reset")

In a secure communication environment (i.e., A1, and A2 hold) the sending and receiving of "reset" events are well synchronized. Formally: $com(head(sent(X_i \,||\, a(\text{-}reset:b)))) = head(received(select_3(Y_j \,||\, b(reset:a),a)))$.

Proof:

Given the two assumptions (A1) and (A2). Furthermore, we denote C_1: *set(sent(X_i))*$\neq\emptyset$, i.e., *a* hasn't sent anything so far and C_2: *set(received(filter*$_3$*(Y_j,a)))*$\neq\emptyset$, i.e., *b* hasn't received any message from *a*. According to definition 5 of "reset," the following four possibilities exist:

1. *Neither C_1 nor C_2 holds.*

   Then X_i ||*a(-reset:b)*$=\emptyset$ and *select*$_3$*(Y_j||b(reset:a))* = *select*$_3$*(filter*$_3$*(Y_j,a))*$=\emptyset$. Obviously, Theorem 1 is true.

2. *C_1 does not hold, but C_2 holds.*

   This directly contradicts assumption (A1). If there was no message sent by communication partner *a*, then there can't be any message received from *a* by *b*.

3. *C_1 hold and C_2 does not hold.*

   This is a direct contradiction to assumption (A1) as well. If there was no message received by *b* from *a*, then there can't be any message sent from *a* to *b*.

4. *C_1 and C_2 hold.*

Let x_l be the last event sent by *a* before executing reset. Due to premise (A2), *head(received(select*$_3$*(Y_j,a)))=com(x_l)* holds. On the side of communication partner *a*, the event x_l and all successive events to x_l are eliminated during the execution of reset. On the side of communication partner *b*, the event *com(x_l)* and all successive events to *com(x_l)* that are exchanged with the communication partner *a* are eliminated during the execution of reset. All events successive to event x_l that are exchanged with different communication partners are preserved.

If set(*sent(X_i ||a(-reset:b)))*$=\emptyset$ holds, then due to premise (A1) set(*received(select*$_3$*(Y_j || b(reset:a),a)))*$=\emptyset$ holds as well. Thus Theorem 1 holds.

Assume *sent(X_i ||a(-reset:b))*$\neq\emptyset$ and *let x_m=head(sent(X_i ||a(-reset:b))* be the last sent event after the execution of "reset." Given precondition (A1) there exists a state on the partner's side such that *com(x_m)*$\in Y_j$ || *b(reset:a)*. Furthermore, in accordance to premise (A2) *com(head(sent(X_i||a(-reset:b)))= head(received(select*$_3$*(Y_j||b(reset:a),a)))* holds.

Definition 6 (“Cancel”)

Let $a,b \in M$ and X_i be the list of events on the side of communication partner a, i.e., $\forall x \in X_i : \pi_2(x) = a$. And let Y_j be the list of events on the side of communication partner b, respectively. Then we define:

$$X_i \parallel a(-cancel : b) \quad := \quad X_0$$

$$Y_j \parallel b(cancel : a) \quad := \quad filter_3(Y_j, a)$$

Explanation: If a communication partner a executes a “cancel”, then he is set back to his initial state with an empty event list X_0. If a communication partner b receives a “cancel” from communication partner a, then all events sent to or received from a by b are eliminated from his event list.

Remark:

According to definition 6 the following holds: Let $k := |sent(X_i)|+1$ be one more than the number of all sending events in the list of events on the side of a, and let $l := |received(filter(Y_j,a))|+1$ be one more than all events that b has received from a, then

$$X_i \parallel a(-cancel : b) \quad = \quad X_i \parallel^k a(-reset : b) \quad = \quad X_0$$

$$Y_j \parallel b(cancel : a) \quad = \quad Y_j \parallel^l b(reset : a) \quad = \quad filter_3(Y_j, a)$$

The execution of “cancel” by a communication partner a can be expressed by means of the event “reset”. Communication partner a executes $a(-reset{:}b)$ for each event sent by him, until there are no events left or only events that are received by a. By executing an additional $a(-reset{:}b)$, a is set to its initial state with empty event list X_0.

The execution of “cancel” on the partner's side can be specified by the means of the event “reset” as well. Communication partner b receives $b(reset{:}a)$ for each event received from a. The remaining events are all either sent from b to a or are messages exchanged with other communication partners different than a. The remaining events sent to a are deleted by the execution of an additional $b(reset{:}a)$.

In the next step we must make sure that “reset” cannot produce insecure states, i.e., we have to specify a transition rule for “reset”.

4.4 Transition rule for "reset"

A state transition from state S_i to state S_{i+1} stimulated by event $t_{i+1}=a(\text{–reset}:b)$ is permitted, $permitted\left(S_i \xrightarrow{t_{i+1}} S_{i+1}\right)$, if the following rule holds:

[Rule 3] Let T_i be the list of events observed by a before the execution of "reset," and let T_{i+1} be the list of events observed by a after the execution of reset, and let $l := |T_{i+1}|$ be the length of list T_{i+1}. Furthermore, let $T := \overline{tail}^{\,l}(T_i)$ be the list of reverted events. Then $t_{i+1}=a(\text{–reset}:b)$ is permitted iff

$$(a \in W_i \cap W_{i+1}) \wedge (\forall 1 \leq j \leq |T|: permitted\left(S_{l+j} \xrightarrow{T[j]} S_{l+j+1}\right))$$

Explanation: According to [Rule 3], a state transition from state S_i to state S_{i+1} stimulated by event $t_{i+1} = a(\text{-}reset{:}b)$ is an allowed state transition if the voter is eligible and has not yet cast his vote, both, before and after, the execution of "reset" ($a \in W_i \cap W_{i+1}$) and all reverted state transitions were permitted ($permitted\left(S_{l+j-1} \xrightarrow{T[j]} S_{l+j}\right)$).

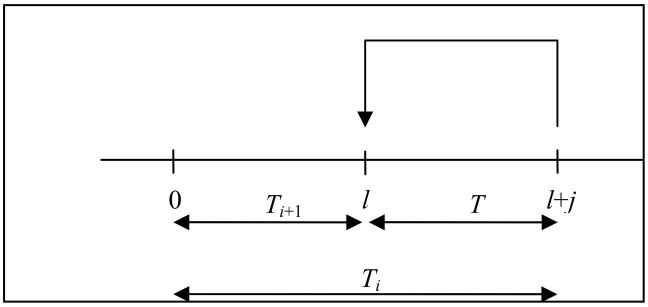

Figure 4.1: Relation between the list of events before and after the execution of "reset."

Remark: [Rule 3] is compatible with both rules, [Rule 1] and [Rule 2], because it resets only permitted transitions. [Rule 3] conforms to [Rule 1] because by the reverted state transitions no vote had been cast into the ballot box. [Rule 3] is compatible with [Rule 2] because the resetting voter would not be one of those voters who had cast votes into the ballot box. Due to the definition of the "reset" function (the filter function in definition 5 makes sure that actions of other participants remain untouched!), the ballots of the other voters would not be reverted, of course.

5 The extended model

In this section, we show that [Rule 3] complies with the security properties (ap.1) and (ap.2) which are equivalent to definition 4.

The specification of an IT security model requires first the specification of secure system states and of permitted state transitions [Gr08]. As a definition for secure system states, we use the definition 4 of section 3.3 above in the version with the two properties (ap.1) and (ap.2), namely that *"voter* is an injective function" (ap.1) and that "$W_{total}=W_i+voter(V_i)$" (ap.2).

Extended security theorem

Permitted state transitions according to [Rule 1] and [Rule 2] of definition 3 as well as according to [Rule 3] from section 4 carry secure states into secure states according to definition 4. Formally, if a state S_i is secure and $permitted\left(S_i \xrightarrow{t_{i+1}} S_{i+1}\right)$, then S_{i+1} is also a secure state.

Proof of the security theorem: For [Rule 1] and [Rule 2] we have proven the security theorem already in section 3. We have only to prove the security theorem with respect to [Rule 3] of secure "resets." To simplify the proof, we first prove the following lemma:

Lemma 1: If a state S_i is secure and $permitted\left(S_{i-1} \xrightarrow{t_i} S_i\right)$, then S_{i-1} was a secure state.

Proof of Lemma 1: If S_i is a secure state and t_i was a permitted state transition, then the state transition t_i was performed according to [Rule 1] or by [Rule2]:

[Rule 1]: Then $V_i= V_{i-1}$ and $W_i= W_{i-1}$ hold. Since S_i is secure, S_{i-1} was secure as well.

[Rule 2]: Then there exists exactly one vote v'' that has been put into the ballot box during state transition t_i such that $V_{i-1} = V_i \setminus \{v''\}$ and $W_{i-1}= W_i \cup \{voter(v'')\}$. It has to be proven that the properties (ap.1) and (ap.2) hold for S_{i-1}.

(ap.1) Firstly, *voter* is injective on V_{i-1} because $V_{i-1} = V_i \setminus\{v''\} \subset V_i$, and *voter* is assumed to be injective on the full V_i already.

(ap.2) Secondly, it must be shown that $W_{i-1}+voter(V_{i-1})=W_{total}$:

(i) $W_{i-1} \cup voter(V_{i-1}) = W_{total}$ holds because *voter* is injective, and therefore $W_{i-1} \cup voter(V_{i-1}) = W_i \cup \{voter(v'')\} \cup voter(V_i \setminus \{v''\}) = W_i \cup \{voter(v'')\} \cup (voter(V_i) \setminus \{voter(v'')\}) = W_i \cup voter(V_i) = W_{total}$. The last equality holds because *Si* is assumed to be secure.

(ii) $W_{i\text{-}1} \cap voter(V_{i\text{-}1}) = \varnothing$ is true because:
$W_{i\text{-}1} \cap voter(V_{i\text{-}1}) = (W_i \cup \{voter(v'')\}) \cap voter(V_i \setminus \{v''\})$. Since S_i is a secure state such that $W_i \cap voter(V_i) = \varnothing$ holds, it is sufficient to prove that $\{voter(v'')\} \cap voter(V_i \setminus \{v''\}) = \varnothing$ holds. And this is true because *voter* is injective.

This completes the proof of Lemma 1.

Given the Lemma 1 above, the proof of the security theorem with respect to [Rule 3] is trivial: If t_{i+1} follows [Rule 3] and S_i was secure, then all reverted state transitions were permitted according to [Rule 3], and hence S_{i+1} is a secure state according to our Lemma 1 above. □

6 Transition rules in a voting process

In the previous sections we have specified conditions for allowed state transitions. In this section we show, at which points in a voting process these rules are to be applied. There are several variants conceivable for each voter's polling process [VV08]. Since we are not going to discuss process designs, we have chosen one process variant with login at start of the voting process.

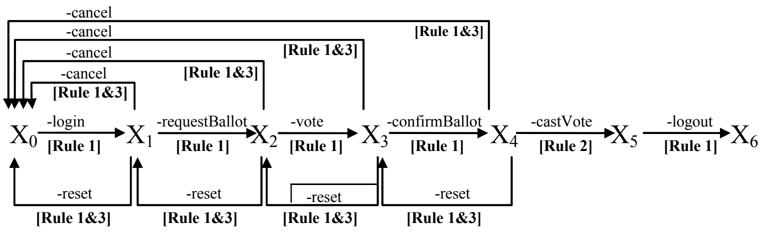

Figure 6.1: Mapping of transition rules on a (simple version of a) voting process

A sequence of transitions of the polling process is exemplarily shown in figure 6.1 where only the client side of the electronic voting process is considered. The voter identifies and authenticates himself by sending his data to the voting server (*-login*). If the voter is unmistakably identified and authenticated on the server's side, the voter is able to request the ballot form (*-requestBallot*). The ballot form is displayed on the voter's client and the voter makes his voting decision (*-vote*). The voter has to confirm his ballot (-*confirmBallot*) to protect against errors by haste. Afterwards he casts a vote into the ballot box (*-castVote*), where the casting of the vote follows [Rule 2]. The voter is allowed to correct his vote (*-reset*) or abort (*-cancel*) his voting process any time prior to the final casting of the vote, where "reset" and "cancel" follow [Rule 1] and [Rule 3].

$$\begin{tikzcd}
X_0 \arrow[r, "\text{-login}"] & X_1 \arrow[r, "\text{-requestBallot}"] & X_2 \arrow[r, "\text{-vote}"] & X_3 \arrow[r, "\text{-confirmBallot}"] & X_4 \arrow[r, "\text{-castVote}"] & X_5 \arrow[r, "\text{-logout}"] \arrow[l, bend left, "\text{-reset}"] & X_6
\end{tikzcd}$$

Figure 6.2: Example of an illegal placing of "reset" in the voting process

But the voter should not be allowed to correct or abort his vote after the final casting of his vote, as shown in figure 6.2. If he could do that, he would obtain the possibility to cast a vote into the ballot box for a second time. Note that our recommendation for the placement of "reset" and "cancel" complies with the security transition [Rule 3] which states that the voter is eligible, both, before and after the execution of "reset" and that all reverted state transitions were permitted.

7 Conclusion

In this paper an IT security model formalizes some basic security requirements for electronic voting: one voter one vote, eligible voters, the correction of a vote, and the abortion of a voting process. The corresponding security properties are specified as secure system states. The voting functions are controlled by state transition rules. We prove mathematically that a function following the rules would transfer a secure state into a secure state.

This contribution demonstrates how security requirements for electronic voting can be formalized and how an existing IT security model can be extended by adding gradually security objectives. However, we have not yet included anonymity or verifiability in our model. For a complete formalization of the security requirements for electronic voting, the IT security model presented in this paper needs to be extended by the remaining security objectives defined in the Protection Profile [VV08] and [GH09] step-by-step. Our next research step is to incorporate voter anonymity.

Bibliography

[Ba06] Bachmann, Gregor: Private Ordnung („Private Regime“). Jus Privatum 112, Mohr Siebeck, Tübingen 2006. Esp. S. 293 on precipitance and legal certainty of promises, also in the Anglo-Saxon legal domain.

[BP73] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations, and A mathematical model. ESD-TR-73-278, MTR-2547, Vols 1&2. The MITRE Corporation, Bedford, MA, Nov 1973.

[CC06] Common Criteria for Information Technology Security Evaluation, and Common Methodology for Information Technology Security Evaluation, Version 3.1, 2006

[CE04] Council of Europe. Legal, operational and technical standards for e-voting. Recommendation rec(2004)11 adopted by the committee of ministers of the Council of Europe and explanatory memorandum. Strassburg, 2004.

[DD85] US Department of Defense: Trusted Computer System Evaluation Criteria, DoD 5200.28-STD, Dec l985, http://csrc.nist.gov/publications/history/dod85.pdf [25 Feb 2010]

[GH09] Grimm, R., Hupf, K.: Sicherheitsanforderungen an Onlinewahlen, In: Pichler (Hrsg.), Österreichischer Workshop über Elektronische Wahlen, Salzburg, Dezember 2009.

[Gr08] Grimm, R.: IT-Sicherheitsmodelle. Technical Report 03/2008, Institut für Wirtschafts- und Verwaltungsinformatik, Universität Koblenz-Landau, 2008

[Gr09] Grimm, R.: A Formal IT-Security Model for a Weak Fair-Exchange Cooperation with Non-Repudiation Proofs. In SECURWARE 2009, The Third International Conference on Emerging Security Information, Systems and Technologies, Athens, 18-23 June 2009. IEEE Computer Society Press, 2009

[MG08] MIT/GNU Scheme 7.7.90+, Chap. 7 Lists, MIT, Boston Massachusetts, 2008, http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/ [25 Feb 2010]

[VG08] Volkamer, M., Grimm, R.: Development of a Formal IT Security Model for Remote Electronic Voting Systems. In Electronic Voting, pages 185-196, 2008.

[VV08] Volkamer, M., Vogt, R.: Common Criteria Protection Profile For Basic Set of Security Requirements for Online Voting Products. BSI-CC-PP-0037, Version 1.0, 18. April 2008. http://www.bsi.bund.de/ [visited Feb 8, 2010]

[Wa05] Wang, Andy Ju An: Information Security Models and Metrics. Proceedings of the 43rd ACM Southeast Regional Conference, Vol 2, Security Session, 2005, pp. 178 - 184.